# HTRC Data API Performance Study

Yiming Sun*, Beth Plale†, Jiaan Zeng†
*Amazon
†Indiana University Bloomington
{plale, jiaazeng}@cs.indiana.edu

***Abstract*—HathiTrust Research Center (HTRC) allows users to access more than 3 million volumes through a service called Data API. Data API plays an important role in HTRC infrastructure. It hides internal complexity from user, protects against malicious or inadvertent damages to data and separates underlying storage solution with interface so that underlying storage may be replaced with better solutions without affecting client code. We carried out extensive evaluations on the HTRC Data API performance over the Spring 2013. Specifically, we evaluated the rate at which data can be retrieved from the Cassandra cluster under different conditions, impact of different compression levels, and HTTP/HTTPS data transfer. The evaluation presents performance aspects of different software pieces in Data API as well as guides us to have optimal settings for Data API.**

***Keywords*-Cassandra; performance;**

## I. Introduction

The Data API is a key component of the HathiTrust Research Center (HTRC) [1] cyberinfrastructure that it allows a client to request and retrieve a large number of full volumes or select pages via a RESTful service interface. The returned volumes or select pages are streamed to the client as a ZIP file, so that the page and/or volume structures are maintained, and data may also be compressed to reduce the size.

The Data API is placed in front of a cluster of NoSQL servers that stores and replicates the volume OCR text. Currently the NoSQL solution we are using is Apache Cassandra [2]. By requesting for data via the Data API instead of directly from the storage cluster, the client programs are spared from having to deal with the complexity of communicating directly to the NoSQL stores, the OCR text are better protected against malicious or inadvertent damages, the client activities can be audited, and the underlying storage solution as well as the Data API layer of abstraction implementation may be replaced with better solutions without affecting client code.

Figure 1 is a high-level architecture and data flow diagram of the Data API. The Data API is implemented as a RESTful web service using the Jersey framework. It is hosted on a production VM server with 4 Intel Xeon X7560 cores each clocked at 2.27GHz and 32GB of memory. A client requests for volumes or pages by sending a list of IDs to the Data API. A request handler is instantiated for the client request, which parses the IDs into entries and adds them to a global queue. At the other end of the queue, a number of asynchronous retrievers take entries off the queue and fetch data from the Cassandra cluster. The fetched data are fed into a ZIP Maker that was instantiated for that client. The ZIP Maker aggregates the data into a ZIP file according to the volume and page to which the data belong, and streams the ZIP file back to the client.

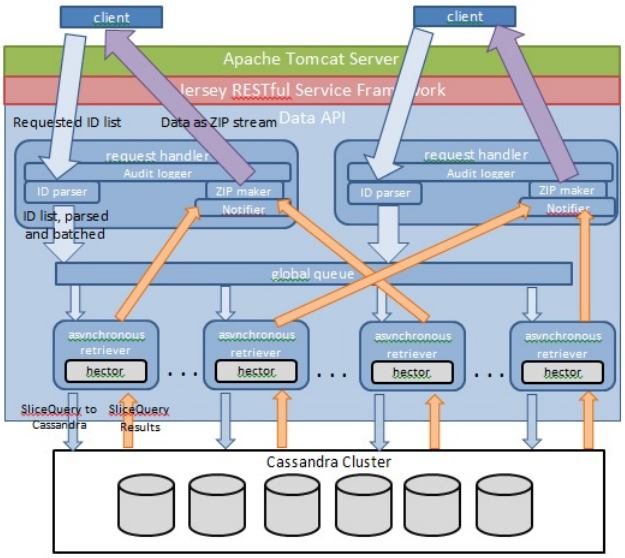


Figure 1. Architecture of Data API

This performance test focuses on the data bandwidth at key points, which are:

1) The rate at which data can be retrieved from the Cassandra cluster
2) The rate at which data can be aggregated into a ZIP stream
3) The rate at which ZIPped data can be retrieved via the Tomcat server
4) The rate at which ZIPped data can be retrieved from a client node

For all parts of this performance test, with the exception of Tomcat [3] HTTP streaming rate, we use a fixed set of 5000 volumes that are pre-selected randomly. The total amount

Work done while Yiming Sun was at Indiana University.

of raw data from these volumes is 4.3GB. For the Tomcat HTTP streaming rate test, we use a fixed set of randomly chosen 500 volumes instead so that the total raw data is about 466MB which can fit entirely into memory.

## II. Retrieving Data from Cassandra

The corpus data is currently stored in a 6-node Cassandra cluster with a replication factor of 2. The nodes that host the Cassandra servers are VM instances on the same VLAN. Each node has 4 Intel Xeon X7560 cores each clocked at 2.27GHz. Five (5) of the nodes have 16GB of physical memory, and one (1) has 32GB. The data storage is a NAS consisting of an array of 15,000RPM SCSI disks totaling 13TB of space and is mounted on every VM instance via NFS. Each physical host is connected to the network via two 10Gbit Ethernet cards. The version of Cassandra we are using is v1.1.1. Currently there are close to 2.6 million volumes in our corpus. The OCR text data is stored in a single Column Family, where each row corresponds to a volume, and the row key is the volume ID. Each page of the volume is a column in that row, but additional columns are added to hold metadata such as the METS file accompanying each volume, copyright information and page count of each volume, as well as byte count and MD5 checksum of each page. While each volume may contain a different number of pages (and thus different number of columns), the majority of the rows are in tens of MBs.

In order to measure the maximum pull rate from Cassandra, we pre-select a random set of 5000 volumes, and fix this set for repeated runs. For each volume, the performance test retrieves only the content of all pages using exactly one request, and in order to do this, the number of pages that each volume has is pre-determined.

Cassandra servers support the optional use of two different caches in memory, a key cache, and a row cache, which can be enabled to speed up the read performance under certain circumstances. The key cache keeps the index of frequently accessed row keys in memory, but access of the columns still goes to the disk; the row cache keeps the data from an entire accessed row in memory so subsequent access to the same row is from memory instead of from disk. If both are enabled, the row cache is checked first for data, and upon a miss, the key cache is then checked (see Figure 2), but typically only one of these caches is enabled. In addition, modern operating systems also utilize free system memory as page cache to keep disk-backed pages for faster access.

We fetch a fixed set of 5000 volumes from Cassandra varying the number of asynchronous fetcher threads (1, 4, 8, 12, and 16). Each scenario is run under different cache setting combinations (the only omitted scenario is enabling both row cache and key cache), and for each run, the performance test takes 5 trials.

[1]Image source: www.datastax.com/wp-content/uploads/2011/04/cache_hits.png

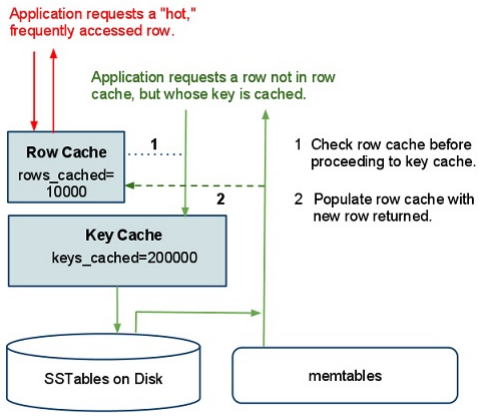


Figure 2. Order in which row cache and key cache are checked.[1]

Figure 3 plots the average data read rate from Cassandra under varying number of asynchronous fetch threads and settings. Each line is a different cache setting combination. From this plot, we can see a clear impact of warm OS page cache. Figure 3 also shows an increase in data read rate with additional asynchronous fetch threads. The impact of OS page cache can also be seen in Figure 4 where the key cache and row cache settings are fixed, and the OS page cache varies between warm to cold using different number of asynchronous fetch threads.

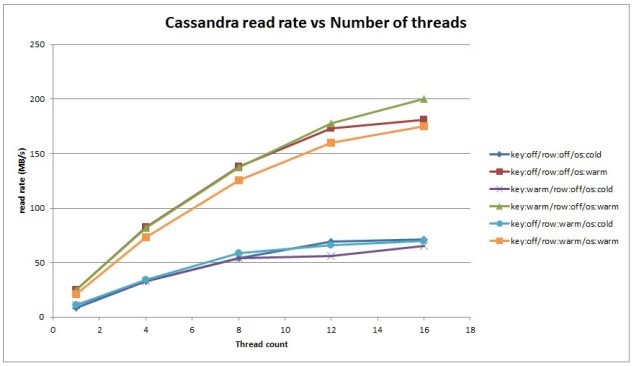


Figure 3. Data read rate from Cassandra with different number of asynchronous threads. Each line is a different cache setting combination.

But on the other hand, the effects of key cache and row cache on data read rate are insignificant, and sometimes, even negative. Figure 5 shows the effect of key cache on the data rate, and Figure 6 shows the effect of row cache on the data rate.

In Cassandra, the key cache is enabled by default. As the data size is about 4.3GB, the caching of row keys in memory does not add significant performance gain. Cassandra ships with row cache disabled by default because row cache costs both memory and CPU, and should only be enabled when

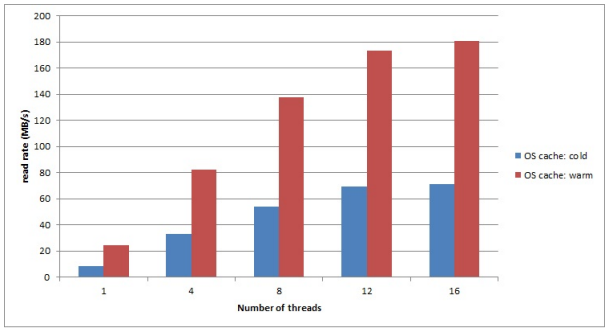


Figure 4. With key cache warm and row cache off, the effect of OS page cache and number of asynchronous fetch threads on the data read rate.

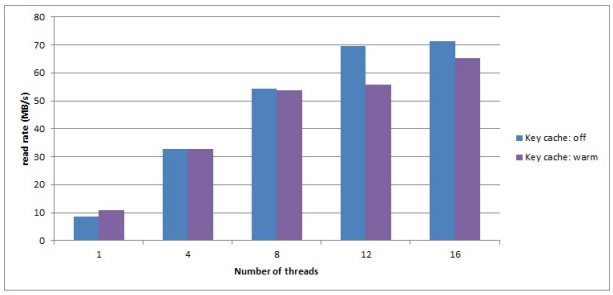


Figure 5. The effect of key cache on the data read rate is insignificant, and for a larger number of threads it has a negative impact. Row cache and OS page cache both disabled.

the usage pattern clearly shows a small set of hot entries that are frequently accessed. Also as row cache stores the entire row in memory, having wider rows also reduces the number of entries a row cache of a given size can hold. Cassandra documentation advises to turn off row cache entirely unless the usage pattern is well understood and shows hot entries.

## III. Performance Test of ZIP Makers

After an asynchronous fetcher thread in the Data API retrieves data from Cassandra, it feeds the data into a ZIP Maker that aggregates the data from different volumes and pages and sent to the client as a ZIP stream.

Depending on the options sent along the request, there are 3 different types of ZIP Maker implementations that each aggregates the data in a different way. The Separate Page ZIP Maker creates a directory entry for each volume, and each page of the same volume is added to the ZIP file as individual text files under the directory. The Combined Page ZIP Maker creates one text file for each volume, and each page of the same volume is appended to the volume text file. The Word Sequence ZIP Maker creates only one text file, and all pages of all volumes are appended into this file. Each ZIP Maker uses the ZIP functionality provided by Java. In ZIP, each entry or file has some additional metadata associated which can be considered overhead in terms of data size. The Separate Page ZIP Maker creates one entry per page, so it has the most overhead; the Combined Page

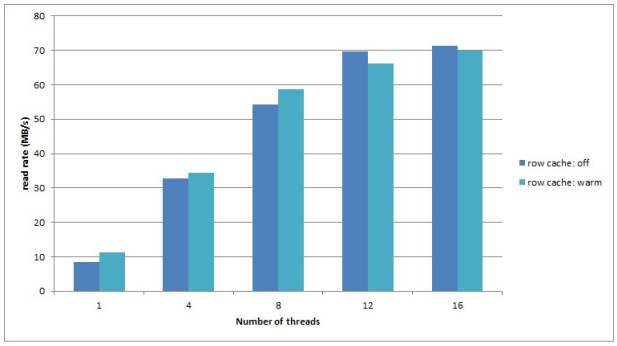


Figure 6. The effect of row cache on the data read rate is insignificant, and at higher thread counts it has a negative effect. Key cache and OS page cache are both disabled.

ZIP Maker creates one entry for volume, so it has fewer overhead; and the Word Sequence ZIP Maker has only one entry, so its overhead is the lowest.

The ZIP functionality in Java allows 10 different levels of compression, where 0 means no compression, and 9 the highest compression. Although the current implementation of the Data API uses 0, for this performance test, we try all compression levels in order to investigate the optimal compression level to use.

For this part of the performance test, the input data is the same set of 5000 volumes mentioned earlier. After all the data is retrieved from Cassandra and stored in memory, the test client feeds the data into a ZIP Maker. All three ZIP Maker implementations are tested, and for each implementation, all 10 compression levels are tested, and 5 trials are taken for each different compression level. The measured metrics include not only the time it takes to perform the compression, but also the data size before and after the compression, so the compression ratio can be calculated and used to find the optimal compression level later.

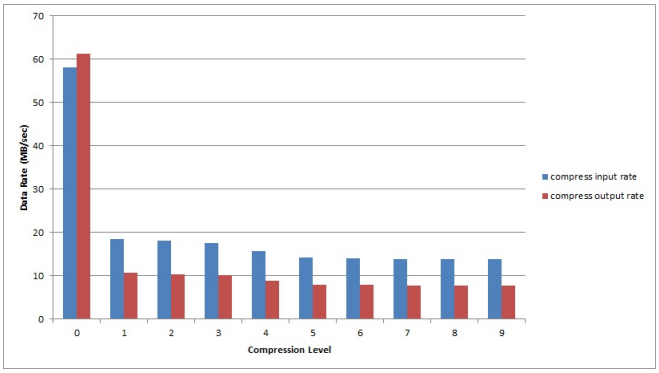


Figure 7. Data input and output rates under different compression levels with Separate Page ZIP Maker.

As the ZIP compression process would take a certain amount of time, we compute the Şinput data rateŤ and

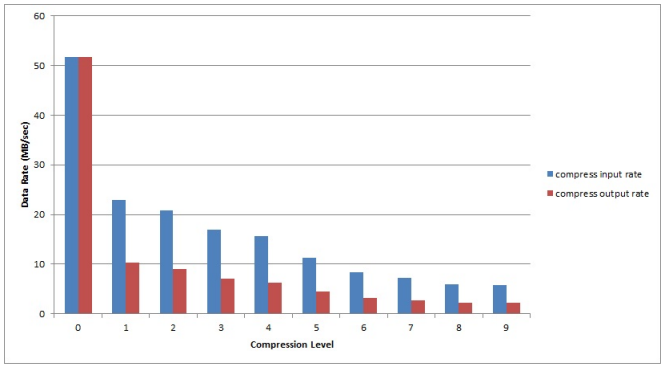


Figure 8. Data input and output rates under different compression levels with Combined Page ZIP Maker.

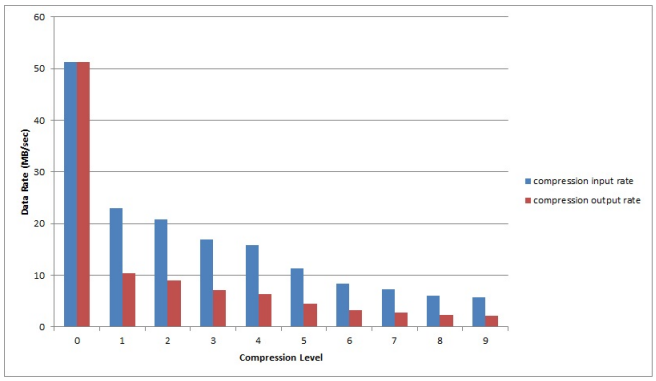


Figure 9. Data input and output rates under different compression levels with Word Sequence ZIP Maker.

Şoutput data rateŤ, where the input data rate is the size of the original data divide by the amount of time, and the output data rate the size of the compressed data divide by the amount of time. Figures 7, 8, and 9 plot the input and output data rates under different compress levels for each ZIP Maker. One observation is that as soon as compression is used (i.e. compression level other than 0), the data rate drops dramatically. Another interesting observation is that for Separate Page ZIP Maker, at compression level 0, the output data rate appears higher than the input data rate Ű this is an artifact of the additional metadata added for all the entries, and with no compression, the output data size end up larger than the input data size. This is actually true for the other two ZIP makers as well, but just not as significant. This is also easier to see using the compression ratio, which is defined as

$$ratio = \frac{size_{compressed}}{size_{original}} \tag{1}$$

Table I lists the compression ratios under different compress levels with each ZIP Maker. From this table, we can see that the lowest compression level of 1 is able to reduce the data size to about ¡ of the original, but higher compression levels only add marginal reductions. Compare the compression ratio to the compress output rate for the same ZIP Maker at the same compress level, the drop in data output rate is more dramatic. Also the compression ratio of the Separate Page ZIP Maker at compress level 0 is clearly greater than 1.0, and that is because of the additional metadata added in the output. Because of the relatively large number of entry metadata in this ZIP Maker, it compression ratio is also higher for each compression level than the other two ZIP Makers.

Compression is used in many data stores internally to reduce the data size and thus reduce the cost of data transmission, so the Data API may also benefit from employing some compression. To determine the right level of compression to use, we must consider the time it takes to compress the data, as well as the time it takes to transfer the compressed data. So the basic idea is:

$$T_{comp} + T_{xc} < T_{xo} \tag{2}$$

where $T_{comp}$ is the time it takes to compress the original data, $T_{xc}$ is the time it takes to transfer the compressed data, and $T_{xo}$ is the time it would take to transfer the uncompressed data. To make the compression worth a while, it should be that the time spent on compression and the time spent transferring compressed data is less than the time it would take to transfer the uncompressed data.

The time it takes to compress the data depends on the size of the original data ($size_{original}$) and the compressor data input rate ($rate_{input}$):

$$T_{comp} = \frac{size_{original}}{rate_{input}} \tag{3}$$

The time it takes to transfer the compressed data depends on the data transfer rate ($rate_{xfer}$) and compressed data size, and the compressed data size in turn depends on the original data size ($size_{original}$) and the compression ratio ($ratio$):

$$T_{xc} = \frac{size_{original} \times ratio}{rate_{xfer}} \tag{4}$$

The time it takes to transfer the original data depends on the data transfer rate ($rate_{xfer}$) and the original data size ($size_{original}$):

$$T_{xo} = \frac{size_{original}}{rate_{xfer}} \tag{5}$$

Substitute these back to the original inequality, we get:

$$\frac{size_{original}}{rate_{input}} + \frac{size_{original} \times ratio}{rate_{xfer}} < \frac{size_{original}}{rate_{xfer}} \tag{6}$$

Simplify the inequality, we get:

$$ratio < 1 - \frac{rate_{xfer}}{rate_{input}} \quad (7)$$

Since the compression ratio cannot be negative, ratexfer/rateinput must be less than 1, and therefore $rate_{xfer} < rateinput$. So in other words, compression should be used when the data transfer speed is slower than that of data compression.

## IV. Performance Test of HTTP/HTTPS Data Transfer

The ZIP stream is transferred to the client via HTTP/HTTPS. The current configuration of the Data API uses HTTPS for security reasons, but the performance test also tests the data transfer via HTTP in order to understand how much drop in data transfer rate HTTPS can incur.

For this performance test, a set of 500 volumes are randomly selected before-hand, and the data is fetched from Cassandra and aggregated into an actual ZIP file on the disk. The size of the ZIP file is 460MB. A special simple RESTful service is deployed into a Tomcat server with both HTTP and HTTPS channels enabled, and this service loads the entire ZIP file into memory as a byte array upon startup, so that when a client makes request to this service, it streams out the data as fast as it can without incurring disk I/O costs.

The service is deployed on the same server (silvermaple) that the Data API is usually hosted, and a test client is run from several different nodes to retrieve the stream from this service, via both the HTTP and HTTPS channels, and it runs 5 trials for each channel from each node.

Figure 10 plots the data transfer rate from silvermaple to these different hosts, via both HTTP and HTTPS. The first data point on the graph is a data transfer from silvermaple to itself (through its real network interface, not the loopback interface), which should be free of most network conditions such as latency or congestion, and this gives us an upper bound on the data transfer rate. As we can see, while the transfer rate via HTTP is able to reach above 500MB/sec, that via HTTPS is only able to reach over 70MB/sec. This shows how expensive HTTPS is. The second data point is taken from ginkgo, which is another VM very similar to silvermaple and also part of the Cassandra cluster. This VM is on the same VLAN as silvermaple, but with only 16GB of memory. While ginkgo is only 1 hop further, its transfer rate via HTTP drops dramatically to only 237MB/sec, which is more than 50% of decrease, but the transfer rate via HTTPS drops to almost 54MB/sec, which is less than 25% of decrease.

The third data point is taken from thatchpalm, one of the two physical machines we use to run some VMs for our own research purpose. These machines are in the same subdomain as our Cassandra VMs, and are also housed in the same data center, and only 2 hops away. The rate via HTTP

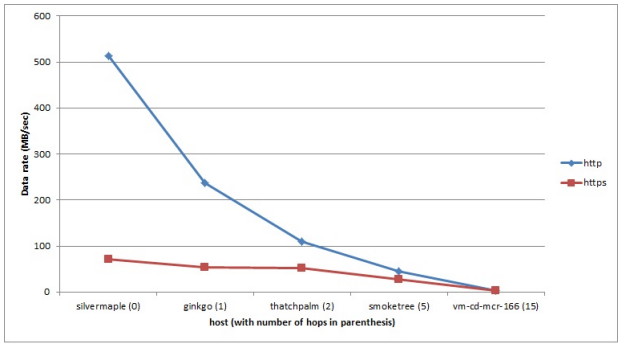


Figure 10. Data transfer rate via HTTP and HTTPS from silvermaple to other hosts.

is further reduced to only about 109MB/sec, whereas the rate via HTTPS is only slightly reduced to about 52MB/sec.

The fourth data point is taken from smoketree, a relatively powerful physical machine we use to do tests. This machine is housed in a different machine room near the opposite corner of our campus, and is 5 hops away. Quite surprisingly, the rate via HTTP seen from this machine is merely 44MB/sec, and the rate via HTTPS is only 27MB/sec. Smoketree is connected to the campus backbone via a 1Gbit link, and the same link is used to serve other servers from the School of Informatics, so the network traffic must be the reason for the drop in the rate.

The last data point is taken from a rather weak VM hosted at UIUC, and it is at least 15 hops away. From this VM, it is only able to get about 2.7MB/sec for both HTTP and HTTPS.

## V. Conclusion

From the Cassandra part of the performance test, we see no immediate benefit from using CassandraŠs built-in key cache and row cache. Instead, the OS page cache works much better if certain entries are repeatedly requested. But as we gather more usage data, we may be able to identify a small set of commonly requested volumes, and if this happens, we can investigate the optimal configuration for Cassandra row caches.

For the Data API, we should keep the number of asynchronous fetch threads to somewhere between 12 to 16 to get the optimal performance, which may deliver a data rate at about 60MB/sec with cold OS cache, and up to 180MB/sec if data is in OS cache. From the ZIP part of the performance test, we can see that compression can reduce data size close to half, but also reduces the data flow rate by more than half. We can also observe that with compression enabled, it is more efficient to compress larger chunks than smaller chunks at lower compression levels, but less efficient at higher compression levels. In other words, when compressing larger chunks, the efficiency decreases more as compression level increases; and when compressing smaller chunks, although

Table I
COMPRESSION RATIOS UNDER DIFFERENT COMPRESS LEVELS.

| Compression level | Combine Page | Separate Page | Word Sequence |
|---|---|---|---|
| 0 | 1.00028179 | 1.055376056 | 1.000152877 |
| 1 | 0.45224775 | 0.581561624 | 0.451995985 |
| 2 | 0.434348842 | 0.576302962 | 0.434015044 |
| 3 | 0.417780772 | 0.573168187 | 0.417371745 |
| 4 | 0.403689643 | 0.56296468 | 0.403398508 |
| 5 | 0.389947053 | 0.559575205 | 0.389606485 |
| 6 | 0.384062183 | 0.559287559 | 0.383691906 |
| 7 | 0.383005471 | 0.559235547 | 0.382628799 |
| 8 | 0.38240618 | 0.559214015 | 0.382034387 |
| 9 | 0.382341142 | 0.559213913 | 0.381967231 |

still on a decreasing trend, the efficiency remains relatively stable as compression level increases.

From the HTTP/HTTPS streaming part of the performance test, we can see that data transfer via HTTPS is much more expensive than that via HTTP because of the overhead in encrypting the data. In addition, we also have made some interesting observation in the decrease of data transfer rate as the receiving host gets further away from the service host. Of course, we are fully aware that the hardware specs of the different hosts vary greatly, and the number of hops is not the only thing that can affect the data transfer rate. It would be ideal to use hosts with similar hardware specs, but for this performance test, we are only able to use what we have access to.

The data transfer rate seen on ginkgo is probably the most accurate representation of data flowing through the VM's NIC. The physical hosts that host these VMs are said to have 2 10Gbit Ethernet cards to the outside world, so getting 259MB/sec via HTTP is fine, since there are other more limiting choke points in the Data API and the transfer channel, and we are probably going to use only HTTPS, which would cut down the transfer rate to 54MB/sec. In addition, because the NICs are shared by all VMs on the same host, other VMs may incur high network I/O traffics that would ultimately affect the Data API's transfer rate.

For all practical purpose, the host that is most likely to be a real-world client machine to the Data API is thatchpalm as it and another similar physical machine are used to carry out our research in supporting non-consumptive text analysis and data mining. So therefore, the more realistic data transfer rate we can expect is about 109MB/sec via HTTP, and 52MB/sec via HTTPS.

## REFERENCES


[1] *HathiTrust Research Center*, http://www.hathitrust.org/htrc.

[2] *Apache Cassandra*, http://cassandra.apache.org.

[3] *Tomcat*, http://tomcat.apache.org/.